Design Patterns in Java

Design Patterns are reusable solutions to common problems in software design. They provide proven approaches to solve recurring design issues, making the software more flexible, reusable, and maintainable. Design patterns in Java fall into three broad categories: **Creational**, **Structural**, and **Behavioral** patterns.

1. Creational Design Patterns

These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

- **Singleton**: Ensures a class has only one instance and provides a global point of access to it.
    - **Example**: Database connection pool, logging, configuration settings.
- **Factory Method**: Defines an interface for creating objects but allows subclasses to alter the type of objects that will be created.
    - **Example**: Creating different types of shapes (circle, square, etc.) in a drawing application.
- **Abstract Factory**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
    - **Example**: Creating UI components for different operating systems (Windows, MacOS, Linux).
- **Builder**: Builds complex objects step by step, separating construction and representation.
    - **Example**: Building a House object with multiple parts (walls, doors, windows, etc.).
- **Prototype**: Creates new objects by copying an existing object, known as the prototype.
    - **Example**: Cloning shapes in a drawing application without knowing their concrete class.

2. Structural Design Patterns

These patterns deal with object composition, simplifying the structure by identifying relationships.

- **Adapter**: Converts the interface of a class into another interface the client expects. It allows incompatible classes to work together.
    - **Example**: Adapting a legacy system to work with a new system.
- **Bridge**: Decouples an abstraction from its implementation, allowing them to vary independently.
    - **Example**: Separating the logic for rendering shapes from their platform-specific rendering details (like different APIs for Windows, MacOS).
- **Composite**: Composes objects into tree structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly.
    - **Example**: File system structure where files and folders are treated similarly.

- **Decorator**: Attaches additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality.
    - **Example**: Adding functionalities like scrolling, borders, or shadows to a window component.
- **Facade**: Provides a simplified interface to a complex system of classes, making it easier to use.
    - **Example**: A DatabaseHelper class that simplifies database operations like opening/closing connections, querying, etc.
- **Flyweight**: Reduces the memory usage by sharing as much data as possible with similar objects.
    - **Example**: Reusing graphical objects like trees in a game to save memory.
- **Proxy**: Provides a placeholder or surrogate for another object to control access to it.
    - **Example**: A virtual proxy that delays the creation of a resource-heavy object until it's needed.

3. Behavioral Design Patterns

These patterns focus on communication between objects, defining how objects interact and how responsibilities are distributed among them.

- **Chain of Responsibility**: Passes a request along a chain of handlers, where each handler either handles the request or passes it to the next one in the chain.
    - **Example**: A support system where requests are escalated from lower-tier to higher-tier personnel if unresolved.
- **Command**: Encapsulates a request as an object, thereby allowing parameterization of clients with different requests, and supports undoable operations.
    - **Example**: A text editor that supports undo/redo operations.
- **Interpreter**: Defines a grammar for interpreting sentences in a language, and provides an interpreter to evaluate the sentences.
    - **Example**: Evaluating mathematical expressions or simple language processing.
- **Iterator**: Provides a way to sequentially access elements of a collection without exposing its underlying representation.
    - **Example**: Iterating over elements in a List or Set in Java.
- **Mediator**: Defines an object that encapsulates how a set of objects interact. It promotes loose coupling by preventing direct communication between objects.
    - **Example**: A chat application where a Mediator object controls communication between different users (participants).
- **Memento**: Captures and externalizes an object’s internal state without violating encapsulation so that the object can be restored to this state later.

    - **Example**: Implementing undo functionality in a text editor.

- **Observer**: Defines a one-to-many relationship between objects so that when one object changes state, all its dependents are notified and updated automatically.
    - **Example**: A notification system where a subject (like a blog) notifies all subscribers (observers) of new updates.
- **State**: Allows an object to alter its behavior when its internal state changes. It appears as if the object has changed its class.
    - **Example**: A Document object that behaves differently when in draft, reviewed, or published state.
- **Strategy**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. The algorithm can vary independently from the clients that use it.
    - **Example**: Different sorting algorithms (quick sort, merge sort) can be applied to a list based on context.
- **Template Method**: Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of the algorithm without changing its structure.
    - **Example**: A framework for generating reports where the overall process is defined, but the content or format can be customized by subclasses.
- **Visitor**: Allows adding further operations to objects without modifying their classes. It separates an algorithm from the object structure it operates on.
    - **Example**: Performing operations on elements of an object structure, like calculating taxes or rendering elements.

Summary of Design Patterns Categories

Category	**Purpose**	**Example Patterns**
Creational	Deal with object creation mechanisms, improving flexibility and reuse of existing code	Singleton, Factory Method, Builder, Prototype
Structural	Simplify the structure by identifying relationships between entities and organizing them	Adapter, Composite, Facade, Proxy, Decorator
Behavioral	Concerned with communication between objects, describing the interaction and responsibility flow	Observer, Strategy, Command, State, Iterator

Design patterns are essential tools for building robust, scalable, and maintainable Java applications, offering pre-tested and standardized solutions to common design issues.